**Relazione Progetto Compilatori 20/21**

Gruppo: Nicola Levorato 1241689, Marco Alessio 1242412

**1. Progettazione del linguaggio**

# Per lo svolgimento del progetto è necessario ideare e definire un linguaggio di programmazione e sviluppare un compilatore che traduca programmi per tale linguaggio in codice 3AC. È richiesto che esso supporti:

* Istruzioni di dichiarazione di variabili, esclusivamente di tipo int.
* Istruzioni di assegnazione di variabili.
* Espressioni aritmetiche e logiche, contenenti sia letterali che variabili.
* Istruzioni if, if-else e while.
* Istruzione di stampa a console del valore di una variabile.

Per definire la sintassi del linguaggio, abbiamo scelto di prendere come riferimento il linguaggio C, introducendovi però alcune modifiche:

* Nonostante sia possibile dichiarare solamente variabili di tipo int, abbiamo deciso di considerare esplicitamente espressioni di tipo booleano, in modo similare a Java. Una espressione booleana può essere ottenuta usando le parole chiave true e false oppure un operatore di confronto tra due espressioni intere, e possono essere combinate tra loro usando operatori logici. Le espressioni booleane non possono essere assegnate ad una variabile intera, vengono impiegate nelle condizioni delle istruzioni if e while, e possono essere argomento della funzione print().
* Esistono due parole chiave true e false per rappresentare rispettivamente i due valori booleani vero e falso.
* Gli operatori logici &&, || e ! sono rappresentati rispettivamente dai token and, or e not.
* Per stampare in output variabili, espressioni intere e booleane, esiste la funzione print() invece di usare una funzione di libreria come printf().
* È possibile usare un singolo operatore di assegnazione o di assegnamento con operazione in una istruzione. Ciò comporta che istruzioni C valide come x = y = 0; oppure

x += y \*= 5; non sono consentite nel nostro linguaggio e devono essere separate in più istruzioni equivalenti.

**2. Implementazione del compilatore**

Per implementare l’analizzatore lessicale del nostro compilatore abbiamo usato Flex, mentre per l’analizzatore sintattico Bison.

Tra i token riconosciuti dall’analizzatore lessicale, oltre a quelli “standard”, abbiamo incluso l’operatore di resto della divisione intera (%), gli operatori di assegnazione con operazione (+=, -=, %=, …), gli operatori di incremento e decremento unario (++ e --), gli operatori di shift (<< e >>) e l’operatore di complemento unario (~). Inoltre, vengono riconosciuti ed ignorati i commenti sia a linea singola (// …) che multilinea (/\* … \*/).

Nell’implementazione dell’analizzatore sintattico, il tipo associato a tutti i token e a tutti i simboli non terminali è il medesimo, la *struct* “address”. In questo modo la loro gestione in Bison si semplifica notevolmente, potendo assegnare un valore solamente a quei campi di interesse per il specifico token o simbolo non terminale della grammatica ed ignorando i rimanenti. La *struct* “address” è così definita:

* addr: contiene il nome della variabile o della variabile temporanea in cui è memorizzato il valore dell’espressione considerata.
* type: contiene il nome del tipo di dato considerato. Viene usato esclusivamente nell’istruzione di dichiarazione di variabili e può assumere esclusivamente il valore int nel nostro progetto.
* next\_label: contiene il nome di una etichetta che punta alla prima istruzione che segue l’istruzione if o while corrente.
* true\_label: contiene il nome di una etichetta che punta alle istruzioni da eseguire qualora la condizione di una istruzione if o while sia vera.
* false\_label: contiene il nome di una etichetta che punta alle istruzioni da eseguire se la condizione di una istruzione if sia falsa.
* begin\_label: contiene il nome di una etichetta che punta alle istruzioni da eseguire per valutare la condizione di una istruzione while.

Per implementare la *symbol table* abbiamo sfruttato l’implementazione *open-source* di una *hash table* in C di Troy D. Hadson (file “uthash.h”) [1]. Per non complicare eccessivamente il nostro progetto, abbiamo deciso di non considerare il ciclo di vita delle variabili; in questo modo una variabile sarà sempre visibile dal punto in cui viene dichiarata fino al termine del programma.

Per gestire la dichiarazione delle variabili e l’assegnazione del tipo corrispondente abbiamo utilizzato la tecnica dello *stack* descritta in classe, anche se l’unico tipo disponibile nel nostro linguaggio è il tipo intero. Questo metodo permette di semplificare la gestione di più tipi di dato, qualora si volesse estendere il linguaggio e renderlo più complesso.

I campi *label* next\_label, true\_label, false\_label e begin\_label vengono generati solamente per le istruzioni che ne fanno uso, come le istruzioni if e while, mentre vengono ignorati per tutte le altre istruzioni. In questo modo si evita di creare etichette inutilmente quando non sono necessarie. La loro gestione avviene tramite il passaggio di attributi ereditati: vengono creati tramite un non terminale fittizio e la loro stampa avviene tramite delle *mid-rule action*, sfruttando inoltre la tecnica dello *stack* vista a lezione.

L’istruzione if è stata così implementata nel nostro compilatore:

1. Viene riconosciuto il token if.
2. Vengono generate le etichette true\_label e false\_label tramite una produzione fittizia.
3. Viene riconosciuto il token(, la condizione booleana e la parentesi chiusa ).
4. Tramite una *mid-action rule* viene generato il codice 3AC per:
   1. Saltare a true\_label se la condizione risulta essere vera.
   2. Saltare a false\_label.
   3. Viene stampata l’etichetta true\_label.
5. Viene riconosciuto il blocco di istruzioni da eseguire per il ramo true dell’if.
6. Se viene riconosciuto il token else:
   1. Viene generato l’etichetta next\_label.
   2. Viene generato il salto a next\_label al fine del blocco di istruzioni del ramo true.
   3. Viene stampata l’etichetta false\_label.
   4. Viene riconosciuto il blocco di istruzioni da eseguire nel ramo false dell’if.

Altrimenti:

* 1. L’etichetta false\_label viene assegnata al campo next\_label.

1. Viene stampata l’etichetta next\_label.

Il passaggio degli attributi ereditati è gestito basandosi su un trucco. Invece di creare l’etichetta next prima di ogni statement questa viene creata direttamente all’inizio di ogni statement che ne ha bisogno come ad esempio l’if, l’if-else e il while. In questo modo si evita di creare etichette inutilmente come per gli statement di dichiarazione o assegnazione di variabile. Poi alla fine di ogni statement viene stampata l’etichetta next creata inizialmente, invece di stamparla una volta aver ridotto lo statement.

Inoltre, tutti gli altri attributi ereditati come true\_label, false\_label e begin\_label insieme a next\_label vengono creati tramite delle mid-rule action o dei non terminali fittizi che eseguono un’azione. La stampa di queste etichette avviene sempre usando dei non terminali fittizi e sfruttando la tecnica dello stack vista in classe.

Inoltre, in uno *statement* di dichiarazione è possibile dichiarare più variabili e opzionalmente inizializzare ciascuna con un valore.

Nell’implementazione dell’analizzatore lessicale abbiamo creato la macro “PROJECT\_LOGGING” con la direttiva #define insieme alla funzione “LOG\_FL” per determinare se abilitare la stampa del logging in output per debuggare più facilmente il codice o per disabilitarla in modo da ottenere in output solamente il three address code generato dal compilatore.

Inoltre, come per l’analizzatore lessicale abbiamo usato la macro “PROJECT\_LOGGING” insieme alla funzione “LOG\_Y” per abilitare o disabilitare la stampa del logging in output.

Il difetto di questa tecnica è che si creano dei salti incondizionati aggiuntivi che potrebbero essere evitati ma, nonostante ciò, il 3AC ottenuto è corretto.

Le maggiori difficoltà riscontrate durante lo sviluppo del compilatore sono state:

1. la gestione degli attributi ereditati: inizialmente abbiamo provato a implementarli usando la SDD presente nelle slide viste a lezione ma abbiamo riscontrato difficoltà riguardo la gestione dello stack;
2. ambiguità dovuta al dangling-else: a causa del fatto che le due istruzioni if e if-else condividono la stessa parte iniziale, se si usano mid-rule action differenti per implementare gli attributi ereditati che sono diversi nei due casi (false\_label nel primo caso coincide con next mentre nel secondo caso viene creato) si verificano dei conflitti reduce/reduce dato che i token sono gli stessi ma bison non sa che azione eseguire. Per risolvere questo problema abbiamo usato un trucco: la prima parte dell’if è la stessa per entrambi i casi quindi next\_label coincide con false\_label, dopo aver processato il token “ELSE” devo assegnare a next\_label una nuova label. Questo è possibile grazie al fatto che nell’if-else la label next viene stampata solo dopo aver processato il token “ELSE”.

**3. Bibliografia**

[1] <https://github.com/troydhanson/uthash>